Chapter 4

Parallel processing implementation

4.1 Introduction

4.1.1 Parallel computing concepts

Parallel processing refers to the concept of speeding up the execution of a program by dividing the program into multiple fragments that can execute simultaneously, each on its own processor. A program being executed across N processors might execute N times faster than it would using a single processor [44]. The term parallelism denotes the possibility of executing several operations (instructions) simultaneously. Nowadays there are a considerable number of results on algorithms and parallel architectures, i.e. algorithms and architectures which make the parallel execution possible [45].

There is an intrinsic parallelism present in many computational problems. While it is sometimes clear that the solution of many problems can be easily obtained by simultaneously executing some stages of the solving process, often a simple conversion of sequential algorithms into parallel algorithms does not provide the maximum attainable parallelism for a given problem. In fact, it is not always possible to transform the best sequential algorithms into the best parallel algorithms, i.e. there is no trivial correspondence between sequential and parallel computing, and the existence of parallel computing environments, opens new horizons for scientific investigation [45].

The first parallel algorithms were developed at the beginning of the sixties, even though parallel architectures did not exist at that time. In that period, and in the following years, many researchers were fascinated by the challenge of solving problems assuming the existence of a parallel environment, without pondering on the applicability of their studies. This research, which later became important in practice, led to the first investigations into the characteristics and properties of parallel computing [45].

4.1.2 Technology

Technology has been growing very rapidly during recent years. Different types of parallel computers have been designed, starting from the vectorial computer, up to the multiprocessor and the distributed system. These new architectures belong to the very large class of parallel computers [45]. The design of parallel machines leads to a strong practical interest towards in the comprehension of all aspects of parallelism, with particular regard to the parallel solution of the main computational problems. Contemporary technologies make it very difficult to connect processors to one another completely and directly [46].

The advent of Very Large Scale Integration (VLSI) techniques has further augmented the interest towards parallelism from a different viewpoint: algorithms are designed in view of their hardware implementation. A very interesting feature of VLSI is its measure of "architectural complexity", i.e. the circuit area. Another attractive characteristic of VLSI is the possibility of having a single circuit at reasonable costs and considerable number of computing elements cooperating for the execution of a given process. Finally, VLSI is also innovative from a theoretical viewpoint: we will see that in order to obtain an appropriate computing model from it, it is necessary to take into account the "geometric" structure of computations, in addition to their logical structure [45].

Recently there has been an increased interest in parallel processing in a Network of Workstations (NOW) as an alternative to Massive Parallel Processors (MPP) due to simplicity and cost effectiveness of such systems.

4.1.3 Classification of parallel machines

The terminology used in parallel computing lacks uniformity even when dealing with the basic issues. This is because a great number of ideas, concepts and methods differ both from a logical and computer architectural point of view when referring to the notion of parallelism. The characteristic which is common to all aspects of parallelism is the concept of concurrence, i.e. the simultaneous participation of entities in the quest towards a common goal [45].

Developing the notion of concurrency implies abandoning both the traditional machine model (Von Neumann computer) and the sequential nature of its algorithms. The problem then arises of characterizing new computing methodologies [45]. Already in 1966, some basic criteria were identified to classify parallel computing from a programmer's point of view. Programming with a single instruction flow was called Single Instruction Multiple Data (SIMD), whereas programming with several instruction flows was called Multiple Instruction Multiple Data (MIMD). The SIMD and MIMD classes proved to be an important means of classification, since almost all parallel machines are actually included in them [45]. SIMD programming is also called synchronous parallel programming. There is a single program and all processors are constrained to execute the same instructions. The parallel flows which are automatically generated by an instruction on different data converge at the end of the instruction execution. In the case of MIMD parallelism, the programmer has the possibility of controlling many instruction (process) flows and many data flows. All processes can be made up of instructions belonging to different programs [45]. This high flexibility makes MIMD

parallelism more general than SIMD parallelism: a problem which is not characterized by a regular structure, but has a potential parallel exploitation, is appropriate for MIMD and not for SIMD. The greater flexibility must be paid for by the presence of problems which do not arise in the case of SIMD, in particular problems of data and process synchronization, as well as problems of allocation of processors to processes [45].

4.1.4 Interconnecting structures and graphs

One of the main obstacles to the design of parallel computers resides in the difficulty of efficiently implementing hardware and/or software communication mechanisms among the different computing elements [45]. If all the elements are connected by a shared resource, this becomes the bottleneck of the system with consequently slower execution and worse performance. Hence, the interconnection problem should be addressed with the adoption of dedicated links.

In this framework, a parallel architecture can be seen as a set of processors connected by links for exchanging messages. In order to analyze the properties of interconnection structures, it is convenient to adopt the notation of a graph. In fact, it is possible to establish a correspondence between the nodes of a graph and computing elements, as well as between arcs and links. The use of this correspondence makes the study of the topological properties of the structure much easier. It follows that if the focus is on the geometry of the connections, a graph is a natural representation of the architecture [45].

4.1.5 Parallel computing models and complexity measures

Computational complexity is a main field of computer science devoted to the determination of the minimal quantity of resources required to solve a computational problem. The investigation is usually carried out in two different ways:

- One method attempts to evaluate the quantity of resources necessary to solve a problem (complexity lower bound).
- The other method analyzes the way algorithms use their resources (complexity upper bound) [45].

If one wants to evaluate algorithm performance and compare it to the best possible performance, it is necessary to introduce computing models highlighting the fundamental resources and their relationships. It is then possible to define an optimum algorithm as the one making the minimum use of resources. The analyses carried out on the models give results that can be applied to the machines [45]. An apparent contradiction regarding the complexity of parallel computations is the fact that many computing models do not consider the costs due to data access and data exchange. Neglecting these costs is in certain cases an excessive simplification of reality. On the other hand, the lower complexity bounds obtained by neglecting these costs are of course still valid, when these costs are present [45]. It follows that the study and the analyses, even if performed on simplified models, give useful indications,

provided that a model expressing some of the particularities of parallel machines is used. For this reason, many complexity analyses are performed on models neglecting some costs [45]. One of the main objectives of computational complexity is to determine bounds to the cost required by computations, independently of technical details. Hence, it is necessary to define how to measure the performance of parallel algorithms independently of the specific architecture. We can introduce some of the main parallel computational models, (Boolean and arithmetic) circuits and parallel random access machines (PRAM), together with an "algorithmic" model, which is less formal than the preceding ones, but more useful to analyze the performance of algorithms. This parallel computing model has been widely used, especially for the early complexity analyses. The following rules describe the criteria to perform an analysis using such a model [45]:

- Any number of processors can be used at any time.
- Each processor can execute one (arithmetic or logic) operation in a unit of time.
- Data access has no cost.
- Communication among processors has no cost.

One of the most important aspects of computational models is their generality. The notion of a computational graph of a parallel algorithm is the unifying element of our approach [45].

In structural analysis, the above assumptions are reasonable if the cost of evaluating a function is orders of magnitude larger than the cost of communication. If the analyses is evaluated using FEM, the former is definitely true.

4.2 Parallel virtual machine, linux xpvm

4.2.1 Linux

> *A quiet rebellion is under way in the world of computer operating systems and software. The rebels refer to themselves as the Open Source Movement. Their doctrine is called the "GNU Manifesto, " a document available from the free software foundation in Boston and their standard is the Linux operating system [47].*

Linux is the free Unix written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers from across the Internet [47]. Linux aims towards POSIX compliance, and has all of the features one would expect of a modern, fully fledged Unix: true multitasking, virtual memory, shared libraries, demand loading, shared, copy-on-write executables, proper memory management and TCP/IP networking [48].

Linux runs mainly on 386/486/586-based PCs, using the hardware facilities of the 80386 processor family (TSS segments, and others) to implement these features [48]. Linux supports GCC, Emacs, the X Window System, all the standard Unix utilities, TCP/IP (including

SLIP and PPP), and all of the numerous programs that people have compiled or ported to it [48]. One might state that UNIX is like Linux. With the freely available source code of the kernel, every user trying to use this own OS (operating system) for a particular purpose is in fact a developer. This tendency causes the use and admiration for Linux to grow daily. Linux is an emerging technology that will affect us all [47].

4.2.2 Parallel virtual machine

Parallel Virtual Machine (PVM) is a software system that enables a collection of heterogeneous[1] computers to be used as a coherent and flexible concurrent computational resource [49]. The individual computers may be shared- or local-memory multiprocessors, vector supercomputers, specialized graphics engines, or scalar workstations, that may be interconnected by a variety of networks, such as Ethernet or FDDI [49]. User programs may be written in C, C++ or FORTRAN and access PVM through library routines [49]. PVM is available as source code and can be installed on any operating system. The libraries handle the inter-communication and the user need only use these libraries for the sending and receiving of data. The programming can be split into two areas, the creation of the master program and slave programs. The master program controls each of the slave programs. Any number of slaves can be spawned on each machine and the results sent back to master after termination. One of the major advantages of PVM is that it uses an already available amount of resources to obtain the goal in a very effective manner. PVM is the most common NOW implementation in use today. The source code of PVM is granted under the GNU license which makes it distributable/available for everyone to use. In fact the PVM software comes as a default installation with most Linux distributions.

Below is a brief description of some aspects of PVM.

Heterogeneity

PVM supports heterogeneity at three levels:

- Applications, machines and networks. At the application level, sub-tasks can exploit the architecture best suited for them.
- At the machine level, computers with different data formats are supported, including serial, vector and parallel architectures.
- The virtual machine can be interconnected via different networks, at the network level.

Under PVM, a user-defined collection of computational resources can be dynamically configured to appear as one large distributed-memory computer, called a "virtual machine"

[1] Quality of being diverse and not comparable

Computing model

PVM supports a straightforward message passing model. Using dedicated tools, one can automatically start up tasks on the virtual machine. A task, in this context, is a unit of computation, analogous to a UNIX process. PVM allows the tasks to communicate and synchronize with each other. By sending and receiving messages, multiple tasks of an application can co-operate to solve a problem in parallel. The model assumes that any task can send a message to any other PVM task, with no limit on the size or amount of the messages [50].

Implementation

PVM is composed of two parts. The first is the library of PVM interface routines. These routines provide a set of primitives to perform invocation and termination of tasks, message transmission and reception, synchronization, broadcasts, mutual exclusion and shared memory. Application programs must be linked with this library to use PVM. The second part consists of supporting software that is executed on all the computers, and make up the virtual machine, called a "daemon". These daemons interconnect with each other through the network. Each daemon is responsible for all the application components processes executing on its host. Thus, control is completely distributed, except for one master daemon.

Two crucial topics arise when discussing implementation issues: inter-process communications (IPC) and process control. These topics are discussed below [50].

Inter-process communications

In PVM different daemons communicate via the network. PVM assumes existence of only unreliable, unsequenced, point-to-point data transfer facilities. Therefore, the required reliability as well as additional operations like broadcasts, are built into PVM, ontop the UDP protocol. For IPC, the data is routed via the daemons, e.g., when task A invokes a send operation, the data is transferred to the local daemon, which decodes the destination host and transfers the data to the destination daemon. This local daemon decodes the destination task and delivers the data. This protocol uses three data transfers, of which one is across the network. Alternatively, a direct-routing policy can be chosen (depending on available resources). In this policy, after the first communication instance between two tasks, the routing data is locally cached (at the task). Subsequent calls are performed directly according to this information. This way, the number of data transfers over the network is reduced to only one, over the network. Additional overheads are incurred by acknowledgment schemes and packing/unpacking operations [50].

Process Control

Process control includes the policies and means by which PVM manages the assignment of tasks to processors and controls their execution. In PVM, the computational resources may be accessed by tasks using four different policies:

- A transparent-mode policy, in which sub-tasks are automatically assigned to available nodes.
- The architecture-dependent mode, in which the assignment policy of PVM is subject to specific architecture constraints.
- The machine-specific mode, in which a particular machine may be specified.
- A user's defined policy that can be "hooked" to PVM. Note that this last policy requires a good knowledge of the PVM internals.

PVM uses the transparent mode policy, by default. In this case, when a task initiation request is invoked, the local daemon determines a candidate pool of target nodes (from the nodes of the virtual machine), and selects the next node from this pool in a round-robin manner. The main implications of this policy are the inability of PVM to distinguish between machines of different speeds, and the fact that PVM ignores the load variations among the different nodes [50].

Construction of a PVM cluster

In this study an 18 node Beowulf "Souper" computer is constructed using PCs that would otherwise have been disposed of. All of the computers used are installed with a Redhat 6.2 Linux distribution. A main server is set up to be in charge of all the major tasks. The server uses NFS (Network file system) to share the user home partitions and most of the user programs over the network. This means that each of the other nodes only needs a basic kernel to allow the rest of the file systems to be mounted over the network. Thus all changes in software only needs to be done on the server. Figure 4.1 shows the cluster constructed.

Installation and configuration of such a "Souper" computer cannot be seen as a trivial task. It is however, a necessary step to achieve a computational goal. The time spent constructing such a cluster in co-operation with the effort required for the parallel implementation of a program makes such implementations only viable if the same program can be re-used frequently for different problems. Once such a computing cluster has been installed, entirely new fields of exploration open up in computational mechanics. The gain in computational capabilities allows more traditional problems to be solved with higher accuracy. However, changes in software and programs require almost permanent administration. Details of the software implementation are best obtained by a study of the parallel programs depicted in Appendix C.

4.2.3 Xpvm

XPVM is a graphical console and monitor for PVM. It provides a graphical interface to the PVM console commands and information, along with several animated views to monitor the execution of PVM programs. These views provide information about the interactions among tasks in a parallel PVM program, to assist in debugging and performance tuning [51].

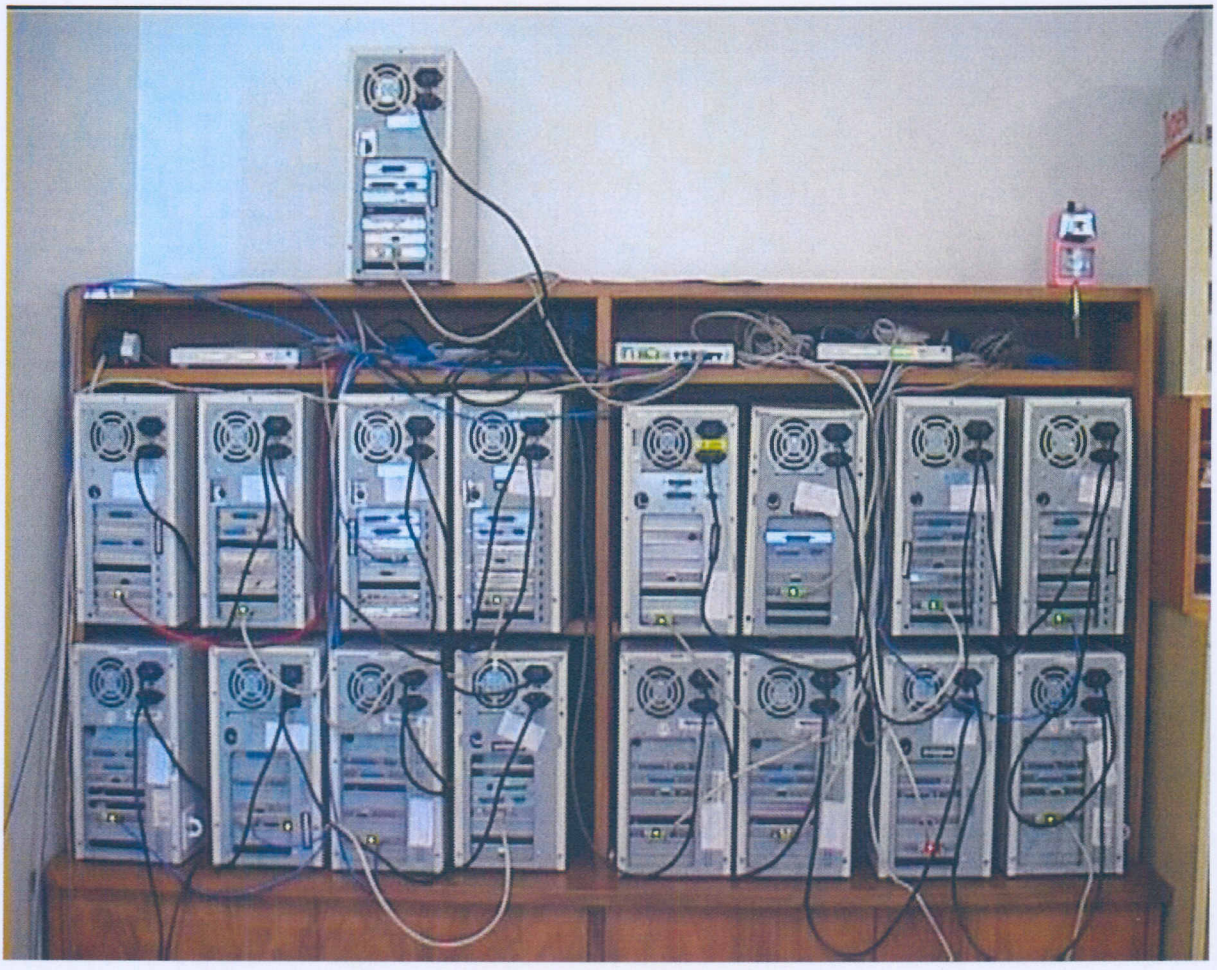

Figure 4.1: Photo of the 18 node Linux "Souper" Computer, named GOH, constructed in this study

To analyze a program using XPVM, a user need only compile his or her program using the PVM library, version 3.3 or later, which has been instrumented to capture tracing information at run-time. Then, any task spawned from XPVM will return trace event information, for analysis in real time, or for post-mortem playback from saved trace files [51]. Figure 4.2 shows a typical XPVM screenshot. However, care should be taken when using XPVM since the monitoring of the PVM program uses a large amount of computer resources which can slow down the execution of the program.

Benchmarking

Benchmarking implies the measuring of the speed with which a computer system will execute a computing task, in a way that will allow comparison between different hardware or software combinations [52]. It does not involve user-friendliness, aesthetic or ergonomic considerations or any other subjective judgment [52].

Benchmarking is a tedious, repetitive task, and requires attention to details. Very often the

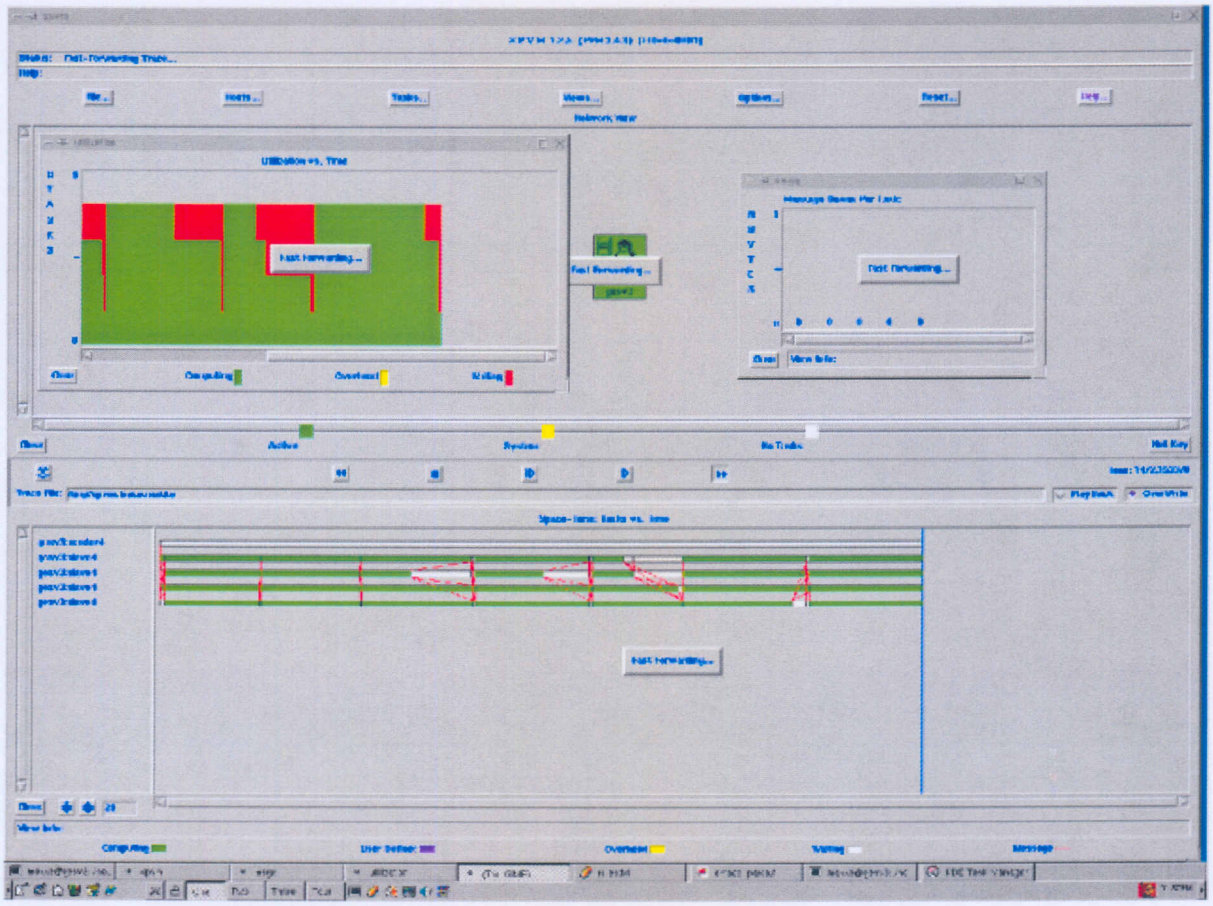

Figure 4.2: XPVM screen shot

results are not what one would expect, and are subject to interpretation (which actually may be the most important part of a benchmarking procedure) [52].

Benchmarking deals with facts and figures, not opinion or approximation [52]. The relative speed of computers can be determined by benchmarking different computer systems against one another for common tasks. When benchmarking programs, they should be run on the same system.

Benchmarking in a parallel environment becomes very intricate since none of the machines in the system can be exactly the same. Many more factors which can influence the speed of the program in a parallel environment than on a single machine and is thus very difficult to benchmark. The interpretation of the results becomes complicated and important. In parallel implementations important factors like data access cost, communication among processors and difference in speed of processors are very difficult too incorporate. Instead the standard idealizations of complexity measures allows the complexity upper and lower bounds to be determined with reasonable accuracy. Determination of complexity bounds gives a possible operational range rather than a specific performance feature for the implementation.

4.3 Parallelization of cellular automata

4.3.1 Parallelization method

Introduction

In this section we proceed to implement a CA program in parallel using PVM. The problem of parallel implementation is approached by first considering the new components required for a parallel implementation. We start with a simple parallel implementation. Doubtless superior implementations exist. At each stage of the parallel implementation our program is evaluated to ascertain possible performance increases obtainable in a PVM implementation.

We have already established that CA computations can become very expensive compared to traditional techniques as the size of the problem increases. Their discrete nature also allows an important analogy with digital computers: CA may be viewed as parallel-processing computers of simple construction [4]. Since each of the cells follow the same cell rule, the parallel architecture becomes a simple implementation of SIMD.

In Section 2.6 we saw that any dimension of CA can be mapped to a square lattice. Thus we only need to consider the implementation of a square computational lattice for a parallel implementation. We also introduced a novel internal computer representation in Section 2.10 which allows for easy interaction with currently available computing architectures.

CA seem to be ideally suited to parallel computations since every cell can be updated simultaneously. This, however, isn't an ideal implementation with conventional parallel computational architectures, since the updating of one cell will require information from more that one neighbor. Thus more data have to be exchanged than the number of computations performed per node. Implementing a parallel version using PVM implies that the data transfer would take place using network cards in each PC. Both 100Mb/s and 10Mb/s network cards will be used. The slower 10Mb/s card will create the biggest bottle neck in system, since the internal data transfer rate in each node (Bus speed) is much higher than the external transfer rate (network card). Additionally the calculation speed of each computer is much higher than the data transfer between its components. Thus, to create an effective parallel program, an attempt has to be made to find an optimum between the amount of data transferred and the actual computations performed. This complex optimum will depend on the number of nodes used in the PVM setup as well as speed difference between the nodes.

The methodology presented here explains how the computing takes place in terms of the physical position of the cell. The difference between this explanation and the internal representation described in Section 2.10 relates purely to indexing and is trivial to implement in such a setup. Since the goal of the implementation of the CA simulation is only to ascertain whether a parallel implementation using PVM would be able to deliver a performance increase, such a representation is excluded in this section.

Parallel implementation data processing

Consider the simple two-dimensional lattice shown in Figure 4.3 with each cell given a number to indicate its position. We obtain the CA boundary conditions from an external method (FEM, BEM or known solutions) and by using initial cell states of zero we obtain an initial computational problem to be calculated with a parallel implementation.

The CA computational lattice can be divided into smaller sub-lattices in which each of the sub-lattices can be computed simultaneously. This means that the boundaries between the sub-lattices have to be exchanged to solve the initial problem. To prescribe boundary conditions for each sub-lattice, fixed boundaries have to be obtained from each neighboring sub-lattice. This results in a complicated setup as the boundaries "overlapping" (as seen in Figures 4.3, 4.4 and 4.5). By creating one "overlapping" boundary row we limit the neighborhood that can be used in our CA simulation to a Moore neighborhood. This does not create any accuracy concerns since the Moore neighborhood has already revealed acceptable accuracy (Section 3.4.2). This neighborhood is also an optimum for the amount of data exchanged.

We divide the lattice into sub-lattices, keeping the boundaries constant on each sub-lattice, to obtain the boundary values from their direct neighbors. If we presume a four sub-lattice split, as shown in Figure 4.4 we need to keep the boundaries of each sub-lattice fixed. Hence the only boundaries that are not updated are the original problem boundaries. This creates three "overlapping" boundaries for each sub-lattice where data has to be exchanged.

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

Figure 4.3: Original square lattice

1	2	3	4
7	8	9	10
13	14	15	16
19	20	21	22

3	4	5	6
9	10	11	12
15	16	17	18
21	22	23	24

13	14	15	16
19	20	21	22
25	26	27	28
31	32	33	34

15	16	17	18
21	22	23	24
27	28	29	30
33	34	35	36

Figure 4.4: Four sub-lattice split

1	2	3
7	8	9
13	14	15

2	3	4	5
8	9	10	11
14	15	16	17

4	5	6
10	11	12
16	17	18

7	8	9
13	14	15
19	20	21
25	26	27

8	9	10	11
14	15	16	17
20	21	22	23
26	27	28	29

10	11	12
16	17	18
22	23	24
28	29	30

19	20	21
25	26	27
31	32	33

20	21	22	23
26	27	28	29
32	33	34	35

22	23	24
28	29	30
34	35	36

Figure 4.5: Nine sub-lattice split

This simple principle can be expanded to more possibilities. We also consider the problem in Figure 4.3 divided into nine sub-lattices. In this case it is clearly not possible to divide the original problem into equally sized sub-lattices. It can, however, be set up as nine computational sub-lattices with different sizes. Figure 4.5 shows one such a possible setup. This example illustrates a few basic logical principles, namely:

- The more sub-lattices that are created, the more overlapping layers are created. This results in more data that have to be exchanged. It also causes the total amount of data in the computation to increase.

- The overlapping layers are not the same for each sub-lattice. The center lattice exchanges data with eight neighbors, the corner lattices exchange data with only three neighbors while the lattices on the sides connect to five neighbors. Obviously, each one connects to different neighbors.
- The different sized sub-problems have different calculation needs. To obtain artificial load balancing between the computing nodes, the more computational intensive problems should be sent to faster computing nodes in the cluster.

4.3.2 Sequential sub-lattice update method

It is clear from the basic description in Figure 4.5 that the implementation of the sub-lattices for computation is not a trivial task. This manipulation of the original data is generally referred to as pre-processing, while combining all the data after processing is referred to as post-processing. Two trains of thought exist regarding parallel computation. The one point of view is that pre- and post-processing of the data do not involve the actual computation and should therefore be neglected when compared to the speed of a single machine. A second viewpoint holds that, since the pre- and post-processing is only necessary when you want to do parallel computations, the cost must be reported when comparing speed.

In order to determine the difference between the two methods and to ensure that the pre- and post-processing is performed well, a program is first written to perform the pre- and post-processing on a single system. By doing this testing, the implementation is simplified, while simultaneously ascertaining the influence. It is also possible to see the influence the pre- and post-processing have on the CA evolution and the total computational time required.

The system used for the bench marking is an Intel PII 300MHz PC running RedHat Linux 6.2. The operating system comes with a utility that is very useful when doing these tests: by combining the binary with the time program through the command "time -v binary", all the data regarding the process can be obtained. For example

```
Command being timed: "t"
User time (seconds): 19.84
System time (seconds): 0.04
Percent of CPU this job got: 96%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:20.64
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 0
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 130
Minor (reclaiming a frame) page faults: 232
Voluntary context switches: 0
Involuntary context switches: 0
```

```
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
```

In this case we are intersted in the user time of the process. The system time reported is the time the process uses to communicate with other parts of the computer. The wall clock time is the total time elapsed since the beginning of the process has started. By using the user time on the same computer, it is easy to see whether or not the pre- and post-processing have any significant influence on the processing time. Table 4.1 shows the time comparisons for the setup used. Some interesting results are obtained. The table shows two columns for each number of cells, the task time in seconds and the number of iterations it takes for the solution to converge.

N sub-	40000 cells		4096 cells		1024 cells		256 cells	
lattices	time (s)	iterations	time (s)	iterations	time (s)	iterations	time (s)	iterations
1	880.39	16789	19.88	3937	1.52	1216	0.10	349
4	854.14	16771	20.36	3894	1.60	1182	0.14	328
9	851.18	16638	21.07	3839	1.74	1159	0.15	315
16	874.17	16770	22.25	3836	1.90	1145	0.18	309
25	877.23	16557	23.29	3836	2.05	1117	0.21	291

Table 4.1: Sequential block update method with 32768 cell states and finite difference rule(N, Number of sub-lattices)

We notice that as the number of sub-lattices increases, the number of iterations required for convergence decreases. To understand why this happens we need to return to Figure 4.5. The first sub-lattice to be updated will be the one in the top left corner. Once this sub-lattice has performed its update, the lattice to the right will start its update, but its left boundary will already contain an updated boundary, and so on. In this manner, with the exception of the first block, all the other blocks will receive either one, two or three updated boundaries which cause the solution to reach a stable state in fewer iterations. This is very similar to the FDM methods of successive approximation to increase convergence rate as discussed in Section A.2.3. Since we are only interested in the final value, it is advantageous to reach the solution in fewer steps.

Consider the example in which we used the greatest number of cells (40000). We note the interesting trend: as mentioned previously the processing time actually decreases slightly and then increases again as the number of sub-lattices increase. The minimum processing time is required at nine sub-lattices. This happens even though the number of iterations actually keeps on decreasing after nine sub-lattices. It can be expected that the processing time is linearly dependent on the amount of iterations as shown in Figure 3.2. This trend illustrates

how the increase in the total amount of data in the system increases the computational need in the parallel methodology.

This implemetation is still extremely ineffective since it combines and splits the lattice after every iteration. But it is sufficient to prove that the effect of pre- and post-processing are negligible in CA computations.

4.3.3 Parallel lattice computation

To introduce real parallel processing we need to expand our implementation to use the PVM libraries to create different programs from our existing code. The first step in using CA in parallel processing is to implement a basic program which can calculate the correct result using separate tasks. The first iteration involves the data being split up and sent to the different processors (slaves) to perform one iteration. The data is then sent back to the master where the data is combined and then split up again. The slaves receive the new problem and perform the calculations. By doing the implementation in this manner we implement the sub-lattice boundary exchanges in a very easy way. This method proved to be very ineffective since the amount of data that has to be transferred between machines is too large for effective computations.

Since the slowest part of the PVM setup is the network connection, the optimum way of improving the performance of the system is to decrease the amount of data that has to be transferred, for a fixed amount of processing. The CA approximation for a linear elastostatic analysis will always have to converge to a fixed state placing the simulation in *class 2*. So a first attempt to improve the ratio is to allow each sub-lattice to converge on its own (local convergence) before boundary exchanges take place. This will reduce the total number of global iterations and thus the number of data exchanges. With the CA solution we are studying, we are only interested in the final state, and the evolution of our CA with local convergence will be different. We only require that this evolution yield the same answer at the final convergence. Table 4.2 indicates that even for a small problem, we obtain a speed-uo factor of roughly 2 in this fashion.

Number of cells	Single iteration		Local convergence	
	time(s)	Iterations	time(s)	Iterations
8×8	34.48	40	16.21	18
16×16	117.81	130	37.81	29

Table 4.2: Local convergence speed improvement for 4 sub-lattices

Although the software created is capable of performing the parallel computation correctly, it is by no means optimized since the data is first sent to the master before being communicated to the correct slave. Although this helps to simplify programming, it does not give any performance gain in the parallel computation. In order to see if parallel computations with PVM would yield any significant performance gain, the parallel implementation has to be modified.

Only the necessary data has to be communicated between slaves and the communication has to be performed with direct slave communication. Implementing a method which would involve inter-slave communication, needs the calculation of the direct neighbors to and from which data has to be sent and received. Each sub-lattice has a maximum of eight neighboring blocks, four sides of which each has to send and receive an array of data and four corner points which is only a single value. Sending and receiving data between the master and the slave programs when one big chunk of data is sent for computation already requires complicated indexing. Allowing each slave to calculate how many different chunks of data had to be sent and received as well as the position of each data set causes complex problems.

To solve this difficulty a flag is set for each data position. This flag is sent with the data so that the receiving slave knows where the amount of data is going to be received and where it had to be placed. This method saves greatly on data transfer since the data is sent directly to the sub-lattice that requires it and not first via the master program. The implementation of such a direct communication is by no means trivial. Although it has the potential to greatly improve performance, it complicates the coding enormously. In fact, the PVM portion of the program became the major part of the program.

Benchmarking on a single machine is already a very controversial issue. It cannot be seen as a simple measurement to compare the performance of a single machine with that of multiple machines with different hardware. All the machines in the cluster consist of different configurations. The only real concept that can be gained from parallel benchmarking is if it is possible to gain a speed improvement through parallel implementation of an algorithm (to gain perspective into the complexity upper bound). It is common in the determination of a systems complexity bounds of a system to use simplified models. It follows that the study and the analysis, even if performed on simplified models, gives useful indications, provided that a model expressing some of the particularities of parallel machines are used [45].

One way in which this can be done on our program in a very idealistic manner is to spawn more than one slave on the same machine and check the user time of each slave (the actual computation time for the slave) on the machine to determine an idealistic time that the process would take on equivalent machines with no external network traffic. The algorithm is tested on various problems and the results are shown in Table 4.3. The speed-up factor is defined as the time used to solve a single problem divided by the time of the multiple sub-lattice problem.

8×8				
Number of sub-lattices	Iterations global	RMS error	Time (s)	Speed-up factor
1	1	1.93	0.006	1.00
4	21	1.93	0.172	0.04
9	26	1.93	0.276	0.02
16	32	1.93	0.463	0.01
16×16				
Number of sub-lattices	Iterations global	RMS error	Time (s)	Speed-up factor
1	1	2.74	0.503	1.00
4	35	2.74	0.296	1.70
9	51	2.74	0.471	1.07

16	58	2.74	0.856	0.59
32×32				
Number of sub-lattices	Iterations global	RMS error	Time (s)	Speed-up factor
1	1	2.99	1.00	
4	51	2.99	0.519	1.92
9	67	2.99	0.689	1.45
16	78	2.99	1.173	0.85
64×64				
Number of sub-lattices	Iterations global	RMS error	Time (s)	Speed-up factor
1	1	2.69	1.503	1.00
4	38	2.69	0.764	1.97
9	49	2.69	0.679	2.21
16	70	2.69	1.231	1.22
128×128				
Number of sub-lattices	Iterations global	RMS error	Time (s)	Speed-up factor
1	1	2.21	8.641	1.00
4	29	2.21	2.543	3.40
9	43	2.21	1.096	7.88
16	33	2.21	0.773	11.18
180×180				
Number of sub-lattices	Iterations global	RMS error	Time (s)	Speed-up factor
1	1	1.98	12.666	1.00
4	37	1.98	6.033	2.10
9	54	1.98	2.556	4.96
16	41	1.98	1.293	9.80
256×256				
Number of sub-lattices	Iterations global	RMS error	Time (s)	Speed-up factor
1	1	1.74	73.418	1.00
4	48	1.74	18.898	3.89
9	58	1.74	6.893	10.65
16	56	1.74	2.965	24.76
400×400				
Number of sub-lattices	Iterations global	RMS error	Time (s)	Speed-up factor
1	1	1.46	339.141	1.00
4	62	1.46	103.431	3.28
9	69	1.46	31.546	10.75
16	73	1.46	10.692	31.72
450×450				
Number of sub-lattices	Iterations global	RMS error	Time (s)	Speed-up factor
1	1	1.39	286.200	1.00
4	66	1.39	107.253	2.67
9	73	1.39	33.712	8.49
16	73	1.46	16.191	17.68
512×512				

Number of sub-lattices	Iterations global	RMS error	Time (s)	Speed-up factor
1	1	1.74	635.810	1.00
4	71	1.74	200.787	3.17
9	77	1.74	63.048	10.08
16	80	1.74	22.309	28.50

Table 4.3: Parallel computing speed performance of CA with PVM

Spawning all the slaves on the same machine, however, does not reflect the true nature of the problem since the network connection between machines which causes the biggest bottle neck in the whole system, is not accounted for. It does, however, give us insight into the approximate complexity upper bound and the evolution of the parallel implementation methodology. We notice that on smaller mesh sizes, there is no gain in parallel processing speed of the solution. We only start seeing a significant speed improvement at a mesh size of 128×128.

For more sub-lattices with bigger mesh sizes, the speed-up factor seems to become greater than the number of sub-lattices.

For a mesh of 256×256 with 16 sub-lattices we obtain a speed improvement of 24. This seems impossible, supposing that we have a process that takes a certain time x on a single processor. It is clear that if we have n of the same processors the same number of computations cannot be performed in less than $\frac{x}{n}$ of the time. In our definition of a speed-up factor for 16 processors we cannot obtain a speed-up factor larger than 16. But on each of our larger problems we obtain values greater than this limit. We should, however, be aware that we are not comparing the same problem. Each sub-lattice division of the problem has a different evolution, although their evolutions lead to the same final state.

Let us consider the factors that cause this phenomenon. The more sub-lattices we have, the more data has to be exchanged per global iteration. The number of global iterations also increases as the amount of sub-lattices increases, since the evolution of the CA is very dependent on the boundary conditions supplied to the problem. Subsequently each sub-lattice obtains boundary conditions at every global iteration, which causes the sub-problem to evolve in a very different way than it would in the single problem. The parallel implementation creates more complicated tendencies as seen in Figure 4.6.

In Figure 4.6 we see that in the first iteration two of the sub-lattices converge very quickly while the other two sub-lattices take much longer to converge. It is clear through the whole simulation that only two of the sub-lattices keep the other processes waiting. (One sub-lattice requires the largest amount of computations.) This process keeps all other processes waiting and thus determines the speed of the whole system. We note the difference in computational requirements for the different sub-lattices.

Running all the processes on the same machine implies that as soon as one slave has stopped processing, the other slaves take up a larger percentage of the central processing unit (CPU). The change in CPU usage causes a small error in the CPU time reported, since the program keeping track of the process only updates at discrete intervals. Investigating the total impact

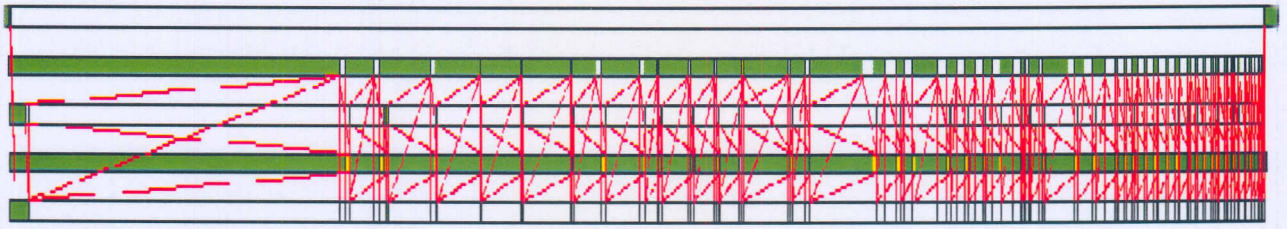

Figure 4.6: Convergence of four sub-lattices (green=processing; white=waiting; red=data communication)

of this error is not possible and also unnecessary, because it only impacts on our already idealized system.

There is a clear pattern that can be observed: as the problem gets closer to convergence, the time between data transfers decreases. By measuring this time one would be able to determine if the problem is converging or diverging. The closer the problem gets to convergence the more frequently data transfer takes place, thereby reducing the computational efficiency of the problem.

4.4 Parallelization of a genetic algorithm

4.4.1 Introduction

Parallelization of a GA enables us to extract cell rules for larger mesh sizes of the CA. Since the function evaluations become very expensive for large mesh sizes, it allows more than one member of the population to be evaluated simultaneously and thus reduces the total time required for the optimization. The implementation of the GA has a further advantage in that the only data that has to be sent over the network is the cell rule and the function value. The fact that only a small amount of data has to be sent forth makes the implementation of the GA in PVM very attractive.

4.4.2 Implementation method

In the CA parallel program, the master is only used to perform the pre- and post-processing of the problem. The master program in the GA performs most of the GA calculations and the slaves are only involved in performing the function evaluations (CA). For big mesh sizes, the function evaluation becomes the most processor intensive part of the calculation. The master initializes each of the slaves (population member) with the whole problem and solution. From this point onwards, the GA (master) performs the usual calculations and sends only the rule to evaluate to each of the slaves. The slave uses the CA rule for the evolution of the problem and once it has converged, compares it to the prescribed solution

and calculates a RMS error which is returned back to the master. This process continues until the maximum number of generations is reached.

Practical Implications

The nature of the problem that is being optimized is such that some CA rules will converge very quickly, some take long, while others may diverge. Since the optimization is performed using discrete variables, there is no way of predicting the outcome and the simulation must be performed. This means that the master program still has to wait for all slaves to finish before it can calculate the values for the next population.

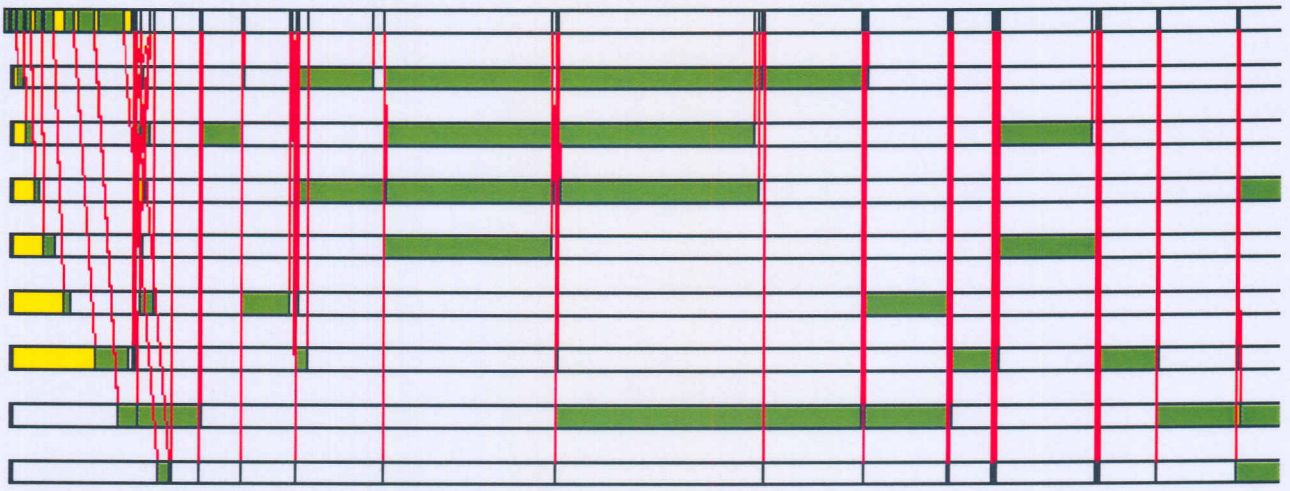

Figure 4.7: Calculation with nine population members (green=processing; white=waiting; red=data communication)

Figure 4.7 shows a typical performance for the parallel GA. Although it is clear that the data transfer is immediate, it is also clear that one or two processes keep the others waiting. To obtain an idea of what possible speed improvement can be obtained, think of the time it takes for the slowest machine on the PVM network to reach the maximum number of iterations that is required for divergence and multiply this time with the number of generations. The result indicates the longest time the PVM program could run. Since the values the GA assigns to the slaves to evaluate is based on random numbers, it is not possible to run a benchmark on the program as was done with the CA. It is, however, clear in finding optimum solutions that this method saves time. By trial and error it is found that the program deliverers relatively good results when approximately three slaves are spawned on the same machine, thus setting the population size to three times the number of machines used. This works relatively well since it rarely happens that more than one slave on the same machine diverges and all the processors are used for almost the whole run.

In actual fact, by changing the number of population members the complexity bounds are adjusted. If we increase the population size we increase the amount of computations required per generation, thereby increasing the complexity lower bound. By not being able to

determine the complexity upper bound and just increasing the lower bound we increase the average of the operational area and thereby gain more performance from the entire system.

4.5 Computational conclusions

The basic principles of, and the gains that can derived from, parallel processing are clear and transparent. However, the determination of optimum configurations in a parallel environment are not so simple. One may in fact consider this problem as complex and quite demanding. The existing technologies that allow for different parallel processing capabilities are diverse, and all subject to different limitations.

In this chapter, the possibilities of parallel processing were investigated using a simple, inexpensive setup, consisting of a cluster of personal computers (PCs) using freely available software. The free standing computers in the cluster contribute processing power to an infrastructure commonly known as a parallel virtual machine (PVM).

While the parallel implementations considered herein are not necessarily optimal, it suffices to demonstrate that a cluster of machines set up from obsolete resources allows for the calculation of solutions faster than is possible on unconnected, free standing resources. Since the cluster can be constructed with no software implementations whatsoever, it makes such a setup ideal for large computational problems when little financial resources are available.

For the parallel implementation of the CA considered in this chapter, a notable performance increase is demonstrated for both the sequential sub-lattice update and the parallel sub-lattice update methods, which are both a direct consequence of the inherent parallelism of CA. This creates the possibility of additional system performance increase by combining these two methods. This combination has the possibility of being quite flexible, since it would allow faster computers to process more than one sub-lattice, while waiting for slower computers to "catch up". The coding involved in such a method is, however, complex and involves the continuous monitoring of each machine in the PVM to allow the master to determine how the calculation is to be distributed over all the available slaves.

It is demonstrated herein that the increase in computational efficiency when simulating CA in parallel does not necessarily yield a monotone increase in gain as the number of sub lattices increases. While this complicates the determination of the optimal division beforehand, it is nevertheless an attractive method to reduce the computational effort associated with machine learning in CA.

In addition, it is shown that parallel implementation of a GA allows for improved solutions in deriving the CA cell rule. Solutions are obtained in a shorter time span, even though the parallel configuration is once again not necessarily optimal.

The parallel GA implementation illustrates the possible benefits of parallel processing in numerical optimization. If the problem (viz. a function evaluation) is evaluated directly by each node, the data transfer between nodes becomes minimal. For computationally expensive function evaluations this results in a fully utilized cluster.

Chapter 5

Concluding Remarks

In this thesis, the suitability of using CA in structural analysis is investigated. In addition, simple approaches for the parallel execution of CA in structural analysis are implemented to evaluate the possible benefits of distributed computing for CA. Machine learning using a distributed genetic algorithm is used to determine the optimum cell rules for the CA, using finite element, boundary element and analytical approximations as the basis for the machine learning.

5.1 Summary of contributions

Firstly, the properties of CA are studied to develop a perspective on the possibilities and limitations of this idealization in structural analysis. The CA is then applied to some simple two-dimensional problems in structural analysis. An efficient formulation for the internal representation of the CA parameters is proposed, which provides for the future modeling of irregular geometries.

Machine learning, similar to the approach proposed by Hajela and Kim, is then used to derive the optimum rules for the CA. It is demonstrated that CA are capable of predicting reasonable approximations to problems in structural analysis, using few discrete elements. The computing cost requirement seems competitive with approximate methods which are currently popular, e.g. the finite element and boundary element method.

However, it is shown that the solutions obtained are not universally applicable, since they vary from problem to problem. Put differently: CA fail to capture the governing equations of structural mechanics, and optimal cell rules have to be determined for each problem studied. Nevertheless, the extraction of the "laws of nature" for a specific problem gives insight in how the CA parameters influence the obtained solutions.

In addition, it is demonstrated that the optimum cell rule is dependent on the number of cell states used in the system. Hence optimal CA cell rules cannot be transferred between meshes of different degrees of refinement for a given problem. Also, the number of cell states used was found to be of lower importance than the total number of cells in the system.

The Moore neighborhood with $r = 1$ is proposed as the optimum cell representation in struc-

tural analysis, since it uses information from all the adjacent neighbors, without requiring an overly complicated boundary cell description. This neighborhood also allows for a simple parallel implementation, resulting in notable speed-up factors.

Furthermore, symmetric problems in structural analysis are analyzed using asymmetric rules in the machine learning process, where the symmetry of the solution found is used as a quantitative indication of the quality of the solution. It is demonstrated that the quality of the asymmetric rules is superior to the quality of symmetric rules, even for those problems that are symmetric in nature.

A parallel computing infrastructure was constructed, by combining 18 obsolete Pentium computers in a single cluster. The software used in assembling the cluster is in the public domain, and is available free of charge.

Finally, exploiting the inherent parallelism of CA, it is shown that distributed computing greatly improves the efficiency of the CA simulation, even though the speed-up factor is not necessarily proportional to the number of sub lattices used.

5.2 Conclusions

While CA are recent additions to the 'tools' used in structural analysis, increased use of CA as distributed computing becomes more widely available is envisaged, even though the CA rules are at this stage not transferable between different problems or even between meshes of varying refinement for a given problem. However, CA remain attractive due to their capability to reveal complex behavior and their ease of interaction on modern computing systems.

CA allow for computations in a discrete system to be performed in parallel in a natural way. By performing computations in parallel on different machines, the speed of execution of a simulation can be improved dramatically. This provides for the continual incease in the maximum size of a structure that can be solved. The implementation on distributed systems is not ideal. However, as advanced technology becomes available that allows thousands of prosesing units in a single computational device, CA will become increasingly suited to parallel computations.

One can even visualize computers designed specifically for CA simulations, based on RISC-like (Reduced Instruction Set Computing) processors. Such "computers" will exhibit great performance gains since they will only require a small instruction set on the CPU itself. Using a small instruction set (the cell rule in CA) will results in extremely fast computers.

5.3 Directions for future work

The following points deserve attention in studies following on this work:

1. Since prediction of good cell rules for CA in structural analysis remains the key issue, increased efficiency and effectiveness in the cell rule learning process is desirable.

2. The use of artificial intelligence in deriving CA cell rules, e.g. using neural networks, is of interest.

3. The efficiency of the simple approach to parallel implementation used herein should be increased.

4. A study of the effect of increased dimensionality on CA simulations is of interest.

5. Finally, CA have been proved to be capable of simulating biological growth. When designing cell rules which allow for material growth in areas of high stress, and conversely, the removal of material in regions of low stress, CA seem a natural candidate for applications in topology optimization.

Bibliography

[1] S. Wolfram. Cellular automaton supercomputing. *High-Speed Computing: Scientific Applications and Algorithm Design*, 1988.

[2] S. Wolfram. Approaches to complexity in engineering. *Physica D*, 22:385–399, October 1986.

[3] S. Wolfram. Cellular automata as models of complexity. *Nature*, pages 419–424, October 1984.

[4] S. Wolfram. Cellular automata. *Los Alamos Science*, 9:2–21, Fall 1983.

[5] S. Wolfram. Satistical mechanics of cellular automata. *Reviews of Modern Physics*, 55:601–644, July 1983.

[6] P. M. Juhl. *The Boundary Element Method for Sound Field Calculations.* PhD thesis, Technical University of Denmark, 1993.

[7] S. Wolfram. Minimal cellular automata approximations to continuum systems. *Originally Presented at Cellular Automata '86*, June 1986.

[8] A. Sameh. Computational science and enigineering. *ACM Computing surveys*, 28(4):810–817, 1996.

[9] G. Spezzano and D. Talia. Programming cellular automata algorithms on parallel computers. *Future generation computing systems*, 16:203–216, 1999.

[10] J. von Neumann. Theory of self reproducing automata. University of Illinois Press, Champaign. IL, 1966.

[11] H. Gutowitz. Frequently asked questions about cellular automata. http://alife.santafe.edu/alife/topics/cas/ca-faq/ca-faq.html, June 1996.

[12] J. R. Weimar. Simulations with cellular automata. http://www.tu-bs.de/institute/WiR/weimar/ZAscript/ZAscript.html, June 1996.

[13] M. Gardner. The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, pages 120–123, October 1970.

[14] M. Gardner. On cellular automata , self-reproduction, the garden of eden and the game "life". *Scientific American*, (224(2)):112–117, February 1971.

[15] Kari Eloranta. Is it alive or is it a cellular automaton?

[16] E. R. Berlekamp, J. H. Conway, and R. K. Guy. Winning ways for your mathematical plays, 1982.

[17] J. C. P. Miller. Periodic forests of stunted trees. *Phlos Trans. R. Soc. London*, A 266:63, 1970.

[18] J. C. P. Miller. Periodic forests of stunted trees. *Phlos Trans. R. Soc. London*, A 293:48, 1980.

[19] H. G. ApSimon. Periodic forests whose largest clearings are size 3. *Proc. R. Soc. London*, A 266:113, 1970.

[20] H. G. ApSimon. Periodic forests whose largest clearings are size ≥ 4. *Proc. R. Soc. London*, A 319:399, 1970.

[21] C. Sutton. Forests and numbers and thinking backwards. *New Science*, 90:209, 1981.

[22] M. Mitchell. An introduction to genetic algorithms. MIT Press, 1996.

[23] A. Scatten. Cellular automata.
http://www.ifs.tuwien.ac.at/ ascatt/info/ca/ca.html, October 1999.

[24] R. Kapral. Discrete models for chemically reacting systems. *Journal of mathematical chemistry*, 6:113–163, 1991.

[25] R. Kapral. Chemical waves and coupled map lattices. *Theory and Applications of Coupled Map Lattices*, pages 135–168, 1993.

[26] S. Wolfram. Univerality and complexity in cellular automata. *Physica D*, pages 1–35, January 1984.

[27] F. S. Beckman. Mathematical foundations of programming, 1980.

[28] G. D. Smith, R. F. Churchhouse, and A. Tayler. *Numerical Solutions of Partial Diffrential Equations: Finite Diffrence Methods.* Oxford University Press, 1985.

[29] H. Abelson and A. A. diSessa. Turtle geometry : The computer as a medium for exploring mathematics. *MIT Press*, 1981.

[30] S. Wolfram, O. Martin, and A. M. Odlyzko. Algebraic properties of cellular automata. *Communications in Mathematical Physics*, 93:219–258, March 1984.

[31] T. Toffoli. Cellular automata mechanics. Technical report, University of Michigan, 1977.

[32] E. Fredkin and T. Toffoli. Conservative logic. *International journal of theoretical physics*, 21:219–253, 1982.

[33] T. Toffoli. Cellular automata as an alternative to (rather than approximation of) differential equation in modeling physics. *Physica D*, 10:117–127, 1984.

[34] J. Hardy, O. De Pazzis, and Y. Pomeau. Molecular dynamics of classical lattice gas. *Physical review A*, 13:1949–1960, 1976.

[35] U. Frish, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. *Physical review letters*, 56:1505–1508, 1986.

[36] Vichniac G. Simulating physics with cellular automata. *Physica D*, 10:96–115, 1984.

[37] C. Bennnett and G. Grinstein. Role of irreversibility in stabilizing complex and nonenergodic behavior in locally interacting discrete systems. *Physical review letters*, 55:657–660, 1985.

[38] P. Hunter and A. Pullan. Fem/bem notes. Department of engineering Science. The University of Auckland New Zealand.

[39] P. Hajela and B. Kim. Ga based learning in cellular automata models for structural analysis. In *Proc. Third World Congress of Structural and Multidisciplinary Optimization*, Niagara Falls, N.Y., U.S.A., May 1999. Paper no. 13-GAM1-2.

[40] R. C. Tirupathi and D. B. Askok. *Introduction to Finite Elements in Engineering.* Prentice Hall, 1997.

[41] J. H. Mathews. *Numerical Methods for Mathematicians, science and engineering.* Prentice Hall International editions, 1992.

[42] J. M. Gere and S. P. Thimoshenko. *Mechanics of Materials.* Chapman Hall, 1991.

[43] V. Apanovitch. *Procision: A tecnical overview.* PhD thesis.

[44] H. Dietz. Linux parallel processing howto. January 1998.

[45] B. Codenotti and M. Leoncini. *Introduction to Parallel Processing.* Addision-Wesley Publishing Company, 1993.

[46] W. Handler. Multiprocessor arrays: Topology, efficiency and fault-tolerance. *Lecture Notes in Computer Science*, 1988.

[47] M. J. Rocchetti. The quiet rebellion. *Mechanical Engineering*, 122(9):86–88, September 2000.

[48] M. Welsh. Linux frequently asked questions with answers. Linux-FAQ.

[49] J. J. Dongarra, G. E. Fagg, G. A. Geist, J. A. Kohl, R. J. Manchek, P. Mucci, P. M. Papadopoulos, S. L. Scott, and V. S. Sunderam. Pvm version 3.4: Parallel virtual machine system. University of Tennessee, Knoxville TN Oak Ridge National Laboratory, Oak Ridge TN Emory University, Atlanta GA.

[50] A. Barak, A. Braverman, A. Gilderman, and O. Laden. Performance of pvm with mosix preemptive process migration scheme. In *7th anual Israeli conference on computer systems and software engineering*, June 1996.

[51] J. Kohl. Xpvm. XPVM Readme.

[52] A.D. Balsa. Linux benchmarking howto. August 1997.

[53] S. P. Timoshenko and J. N. Goodier. *Theory of Elasticity.* McGRAW-HILL Engineering Mechanics Series, 1970.

[54] C.A. Brebbia and J. Dominguez. *Boundary Elements An Introductory Course.* Computational Mechanics Publication, 1989.

[55] Wilson E. Fem class notes.

[56] K. J. Bathe. *Finite Element Procedures.* Prentice Hall, 1996.

[57] Y. W. Kwon and H. Bang. *The Finite Element Method using Matlab.* CRC press, 1997.

[58] S. L. Crouch and A. M. Starfield. *Boundary Element Methods In Solid Mechanics.* George Allen & Uwin (Publishers), 1983.

[59] B. Gucun. Bem for structural analysis. http://cavity.ce.utexas.edu/kinnas/papers/bem/node7.html, 1997.

[60] C. A. Brebbia, J. C. F. Telles, and L. C. Wrobel. Boundary element techniques. *Springer-Verlag Berlin Heidelberg New York*, 1984.

[61] J. Holland. Adaption in natural and artificial systems. Ann Arbor.

[62] C.R. Houck, J.A. Joines, and M.G. Kay. A genetic algorithm for function optimization:aa matlab implementation. Technical report, North Carolina State University.

[63] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, MA, 1989.